GOTO Removal Based On Regular Expressions

Paul Morris
Ronald A. Gray
Robert E. Filman

Software Technology Center
Lockheed Martin Missiles & Space
3251 Hanover Street O/H1–41 B/255
Palo Alto, CA 94304

GOTO Removal Based On Regular Expressions

We present an algorithm for eliminating GOTOs and replacing them with structured if-then-else and loop constructs. Previous approaches have treated GOTO removal as an isolated problem for programming languages. In this paper, we describe a way of reducing GOTO removal to the well-understood problem of converting a finite-state transition network to a regular expression. A semantics is provided showing the regular expression form may be interpreted as a nondeterministic program. A set of pattern-based reduction rules is used to transform the regular expression form back to a conventional structured program. Besides achieving greater conceptual unity, the method leads to a simpler algorithm where the task of recognizing loop boundaries is separated from that of identifying loop exits. We have successfully applied the algorithm in systems for reengineering COBOL/IMS database systems and assembly language code.

ACM Computing Reviews catagories:

D.2.7 Distribution and Maintenance / Restructuring

F.3.3 Studies of Program Constructs / Control primitives

Keywords:

GOTO removal, software reengineering, control structures, restructuring

Introduction and Related Work

Several decades ago, the use of the GOTO statement was a matter of considerable controversy (Dijkstra, 1968). Several papers (Ashcroft *et al.*, 1972, Peterson *et al.*, 1973, Ramshaw, 1988) analyzed GOTOs and proposed algorithms for removing them. In general, the motivation was not so much to retroactively improve existing code as to determine what control structures were needed to render GOTOs unnecessary. Thus, the papers presented conditions for GOTO removal under various standards of equivalence between the transformed program and the original. For example, Ashcroft and Manna (Ashcroft *et al.*, 1972) sought to maintain *functional equivalence*, requiring unchanged input/output behavior, but allowing the introduction of auxiliary variables and statements. Peterson *et al.* (Peterson *et al.*, 1973) were concerned with the higher standard of *path equivalence* in which, roughly speaking, the two programs execute the same sequence of steps. Ramshaw (Ramshaw, 1988) considers an even stronger *structural* standard where the essential program components map to each other in a one-to-one fashion that respects order.

This phase of work on GOTO removal influenced the subsequent design of programming languages, and encouraged a more disciplined approach to coding in general. Today, we see modern languages like Java that include a variety of loop exit mechanisms but lack unstructured direct control transfer.

Our interest in GOTO removal is motivated by the need to renovate legacy systems. Many large software systems developed several decades ago are still in use today. These systems were built using platforms and languages that are increasingly obsolete, and the urgent task exists of upgrading these legacy computer programs to modern standards. Indeed, the situation is regarded in some circles as being in crisis (Gibbs, 1996). One option is to completely rewrite these systems without regard to the existing legacy code. However, this essentially discards the effort that has gone into requirements analysis, design, and debugging over the years. Often, the fruits of this labor reside only in the actual code. A perhaps less wasteful alternative is to reverse-engineer the code to try to extract some or all of this information (Filman, 1996). This may include automatically or manually translating large chunks of code to a more readable high-level form.

Much legacy code is written in assembly, early Fortran, or other low-level languages that do not contain structured control constructs such as WHILE loops and IF-THEN-ELSEs. Instead these programs use unstructured GOTOs for control purposes. Moreover, the accumulated detritus of years of patching often has obscured the flow of control. This has led us to consider the benefit of automatically removing GOTOs in explicating the code. We also had an immediate practical need for a GOTO removal algorithm. Some of our colleagues have developed software, based on dataflow analysis, that is used to generate reports on COBOL/IMS database legacy code for our customers (Polak, 1995). However, the method used requires that the analyzed programs be free of GOTOs. This requirement was violated by several large Cobol programs that were presented for analysis. Thus, our GOTO removal program served as a bridge to extend the range of the software. We have also applied this work to the process of “levitating” IBM 370 assembly language code to higher-level forms (Morris *et al.*, 1996).

Previous work (Ashcroft *et al.*, 1972, Peterson *et al.*, 1973, Ramshaw, 1988, Erosa *et al.*, 1994), while highly developed and abstract in nature, treats GOTO removal as an *isolated* problem in the area of programming languages. This paper is based on the observation that GOTO removal for flowcharts resembles the problem of converting finite-state transition networks to regular expressions, and presents an approach that unifies these two problems. This unification leads to a simpler algorithm. For example, other approaches (Peterson *et al.*, 1973, Ramshaw, 1988) need elaborate machinery for handling overlapping loops, but this is not a difficult issue in the context of generating regular expressions. The complexity of prior methods appears to be due at least in part to the entanglement of the loop recognition task with that of identifying loop exits. In our approach, the two tasks are effectively separated.

Some of the previous methods (Peterson *et al.*, 1973, Ramshaw, 1988) require that subject programs satisfy additional conditions such as reducibility in order to achieve the highest equivalence standard (structural equivalence). Legacy code is in fact almost entirely reducible, but this can not be guaranteed. Moreover, the structural equivalence standard is inappropriate for reverse-engineering purposes, since a simplification is often preferred over maintaining the existing structure. The algorithm presented here is applicable to nonreducible programs, and meets the lesser path-equivalence standard. Subjectively, this standard appears to produce more readable results than approaches that introduce auxiliary variables, such as (Erosa *et al.*, 1994), and is easier to relate to the source presented for restructuring. These considerations are important for reverse-engineering.

While the algorithm discussed in this paper has worked well in practical contexts, we feel its most significant feature is the link to important concepts in computer science, such as finite automata and nondeterminism, and that is our main focus. It is not our purpose to revisit the controversy over the harmfulness of GOTO statements. Our motivation is to pragmatically improve legacy code. We also do not claim computational superiority for our algorithm over others in the literature. It works well in practice, and code is typically reengineered once, not repeatedly. The system has been used in a commercial context to generate code reports on large-scale systems of tens of thousands of lines (Polak *et al.*, 1995).

Overview

In this paper, we describe an algorithm for GOTO removal based on the theory of finite automata (Wulf *et al.*, 1981). It relies on the fact that a finite-state transition network can readily be converted to an equivalent regular expression. Our approach includes translating a computer program into a finite-state transition network, and using a set of pattern-match rules to translate a resulting regular expression back into a structured computer program. This process is illustrated in Figure 1. The regular expression form may be interpreted as a program, but, in order to do so, it is necessary to consider nondeterminism. The final stage of the process restores the program to a conventional form.

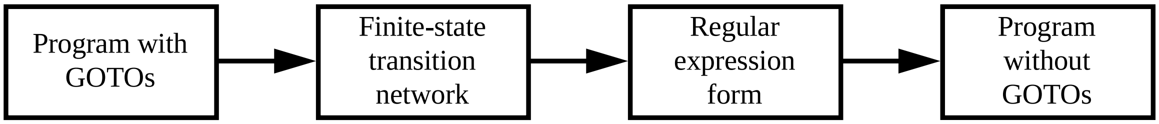


Figure 1. The GOTO-Removal process

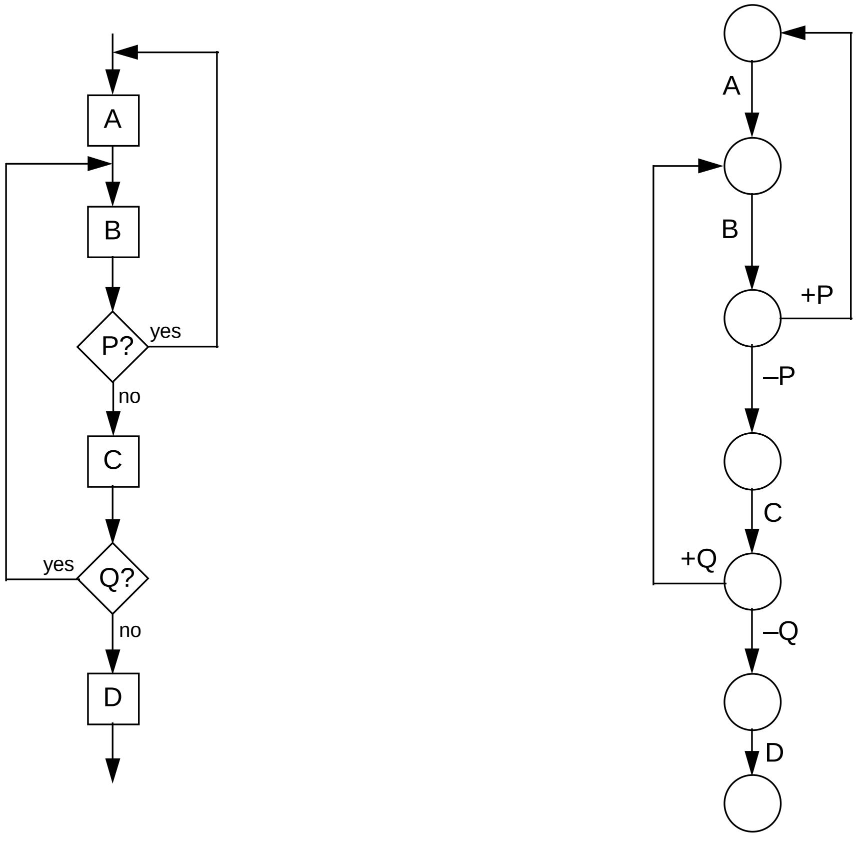
The GOTO-removal system was implemented in Reasoning Systems' Refine tool (Reasoning Systems, 1990). This is a Lisp-based system that provides support for writing parsers to convert a text program into an abstract syntax tree (AST). The nodes in the tree can be decorated with additional information. Refine also incorporates a rule-system that facilitates pattern-based transformation of the AST structures.

In the next section, we discuss certain background concepts underlying this approach. In subsequent sections, we describe the various stages of the GOTO removal process. We then present a detailed example of the algorithm in action. In closing remarks, we summarize the results of this research. In Appendix A, we consider the semantics of the regular expression form, and the issue of nondeterminism. Finally, in Appendix B, we present several before-and-after examples of the algorithm's effect.

Background

Assembly languages permit jumps to locations that may vary dynamically during execution. Except for implementing subroutine returns, use of this feature has generally been considered poor programming practice and rarely occurs in legacy code. Also, legacy programs in assembly and Cobol do not generally make use of “long jumps” that pass through subroutine boundaries. (Even if they do, such jumps can be ignored in a first pass at reverse engineering the code.) In this paper we will ignore the issue of subroutines and only consider programs with GOTOs that jump to fixed locations, i.e., the classic GOTO statement.

More precisely, we only consider programs or program segments that can be represented by means of a *control-flow diagram*, as illustrated in Figure 2(a). Following Peterson *et al.* (Peterson *et al.*, 1973) and other authors, we transform this into an equivalent flowchart, as seen in 2(b). A flowchart (or flowgraph) is a standard syntactic variant of a control-flow diagram where the actions and tests are placed on the arcs following the nodes rather than the nodes themselves. Observe that the arcs of the flowchart are labeled either with tests (negated and unnegated) or statements of the programs. (We have denoted tests in their negated and unnegated forms by prefixing them with - and +, respectively.)



(a) Control-Flow diagram (b) Flowchart

Figure 2. Program Flow Representations

Peterson *et al.* recognize that a flowchart closely resembles a finite-state transition network with a single start and stop node, and use this to define a notion of equivalence for flowcharts in terms of the potential paths through the network, or, equivalently, in terms of the associated regular sets. It should be noted that *potential* here does not necessarily mean *possible*. For example, Peterson *et al.*, state

> Note that if a program has no input data, for example, there is in fact only one path through the flowchart that can actually occur; but in spite of that, we consider the entire usually infinite set of paths that could occur if all possible combinations of test outcomes could occur. ... considering the flowcharts to be transition graphs, ... two flowcharts are equivalent ... if and only if the associated regular sets are equal.

Peterson *et al.* use the associated regular sets merely to define path-equivalence. We take this a step further. In our approach, flowcharts are identified with the corresponding finite-state transition networks. These are used to produce regular expressions as an actual intermediate program representation. The regular expressions are then converted to GOTO-free code in a more conventional program form.

Given the essential role that the associated regular sets play in our approach, the use of a counterfactual ("if all ... combinations ... could occur") in the Peterson *et al.* formulation is unsatisfactory for a formal foundation. For example, it is not *a priori* obvious that programs that are path-equivalent must have the same behavior when executed. The problem is that the transition networks have been introduced as having only a structural relationship to the flowcharts, not a semantic one. In Appendix A, we introduce a semantics for the associated regular sets that fills this gap.

Networks To Regular Expressions

From now on, we regard a flowchart as a finite state transition network with a start and stop node. Recall that a finite state transition network is a representation of a finite automaton. The nodes in the network correspond to states, and the arcs denote possible transitions between states. The arcs are labeled with strings over some abstract alphabet that indicate requirements for the transitions. In the case of a flowchart, the "letters" in the alphabet consist of the conditions and actions that potentially can occur in the program.

It is straight-forward to convert a finite state transition network with a start and stop node to a regular expression. Indeed, Manna (Manna, 1974) presents an algorithm for doing this by systematically eliminating nodes; it thus requires only a linear number of node-removal steps. However, in general, many different but equivalent regular expressions can be formed from the same transition network, and Manna's algorithm provides no assurance that the constructed expression will be a "natural" one, or will not needlessly duplicate subexpressions. Instead of using Manna's approach, we have implemented an algorithm that forms the regular expression in a natural way by computing the paths through the network. This may be summarized as a divide-and-conquer approach that recursively

(1) Computes the elementary nontrivial paths back to the start node.

(2) Computes the paths to the stop node that do not return to the start node.

In (1), the term "elementary" means the start node occurs only at the end-points of the path, and the term "nontrivial" means the path length is greater than zero.

It is easy to see that any path from the start node to the stop node may be built from paths in (1) and (2). More precisely, if the regular expression E_1 represents the result of (1), and E_2 the result of (2), then $E_1^*E_2$ is a regular expression that denotes all the paths through the network. The first set of paths may be empty, i.e., there may be *no* nontrivial path back to the start node. In that case, $E_1^*E_2$ reduces to E_2. Note that if the second set of paths is empty, the entire expression reduces to the empty set of paths.

In the following, it is convenient to use the term “no-path” to refer to the empty set of paths, while “empty-path” refers to the trivial path of zero length.

The algorithm uses a node-marking scheme to prevent revisiting the start node at an interior point of the paths in (1) and (2). The top-level procedure `accumulate-unmarked-paths` is designed to compute the regular expression that would result if all transitions from currently marked nodes were deleted. (Initially, no nodes are marked, but the recursive calls may encounter marked nodes.)

A simplified version of the procedure may be described in pseudocode as follows:

```
procedure accumulate-unmarked-paths (start, stop)
if start=stop then
  return empty-path
else if marked? (start) = true then
  return no-path
else
  begin
    marked? (start) <- true;
    ans <- (merge-successor-unmarked-paths (start, start)) *
            merge-successor-unmarked-paths (start, stop);
    marked? (start) <- false;
    return ans;
  end
end procedure
```

(We remark that, in the implemented system, we actually compute the second call to `accumulate-unmarked-paths` first, since if it evaluates to no-path, there is no need to evaluate the other call.)

The `merge-successor-unmarked-paths` procedure forms regular sub-expressions arising from all the transitions from the start node to its successors, and returns their disjunction. A pseudocode definition would be messy here; instead, we present examples that makes the behavior clear. Suppose the start node has two successors, `node1` and `node2`, reached by transitions labeled with letters `A1` and `A2`, respectively. Then

```
merge-successor-unmarked-paths (start, stop)
```

is given by

```
( A1 accumulate-unmarked-paths (node1, stop)
  | A2 accumulate-unmarked-paths (node2, stop))
```

In the case of a single successor, `node1`, with transition label `A1`, the output would be simply

```
A1 accumulate-unmarked-paths (node1, stop)
```

Note the recursive calls to accumulate-unmarked-paths in each case.

Applying the path algorithm to the flowchart of figure 2(b), we see that the recursive call in the first case (paths back to the start) produces

```
A (B -P C +Q)* B +P
```

while in the second case (paths forward to the end), it produces

```
A (B -P C +Q)*      B -P C -Q      D
```

Note that the cycle beginning with `B` occurs in both subexpressions. Combining the two subexpressions yields

```
(A (B -P C +Q)* B +P)*    A (B -P C +Q)*      B -P C -Q     D
```

for the complete path expression. (Extra spacing is used for readability.)

To improve efficiency in the algorithm, we run a preprocessor that collects certain information. First, the preprocessor determines the *strongly-connected components* (Gibbons, 1985) of the graph, and thus identifies nodes that are not on cycles. Second, it computes the *intervals* (Barrett *et al.*, 1979) of the flowchart. The intervals divide a flowchart into groups of nodes where each group contains a “leader” that *dominates* (always precedes in execution) the other nodes in the group. This enables the preprocessor to identify nodes on cycles that are dominated by other nodes that begin the cycles. The computation of strongly-connected components, and of intervals, uses standard algorithms from the literature that are known to run in linear time.

The information computed by the preprocessor is used to support substantial pruning and caching during the main path-finding process. For example, if the start node is not on a cycle, then the recursive search back to the start node evaluates to no-path, and can be eliminated. Somewhat less obviously, even if the start node is on a cycle, we can still eliminate paths back to it provided it is dominated by some other node that begins the cycle. (The reason for this is that the dominant node will have been encountered first in the recursive descent, and thus will be marked. The mark effectively cuts the cycle, since the path-finding process terminates when it reaches a marked node.) Furthermore, one can cache the regular subexpressions computed for start nodes (with respect to the top-level stop node) that are not on cycles, since each subsequent call with the same start node must return the same value. This avoids the combinatorial explosion that would otherwise result from a sequence of branch-and-join segments of code.

To avoid needless duplication of subexpressions, the algorithm merges identical tails in disjunctive branches as much as possible. The merging does not take place as a separate step, but is integrated into the section of code that constructs the regular expression. Thus, merging is continuously interleaved with the assembly process. Note that the caching described in the previous paragraph may produce shared structure, so in many cases the tails that are being merged are merely different references to the same substructure.

Some duplication in the constructed regular expression is unavoidable. To see this, observe that the flowcharts of regular expressions are always *reducible*. A reducible flowchart (Barrett *et al.*, 1979) is one in which every cycle has only one initial-entry point (where the cycle can be entered for the first time). If the original program had an irreducible flowchart, the construction of the regular expression duplicates some subexpressions, eliminating the irreducibility. In this case, the duplicated subexpressions occur within star expressions that represent the same loop entered at different points. In general, such star expressions are not identical, and therefore will not be merged.

In applications of the algorithm to legacy code, duplication due to nonreducibility did not appear to be a significant issue. However, with one application, we encountered problems due to duplication for a different reason. Consider the following pseudocode description of a program involving a particular if-then-else pattern.

```
      if P then
         A;
         if Q then goto DONE endif;
      endif;
      Z;
DONE: stop;
```

The GOTO removal algorithm will render this essentially as

```
      if P then
           A;
           if not Q then Z; end if;
      else
           Z;
      end if;
      stop;
```

Note that Z is duplicated on both branches of the conditional. Now consider a chain of the form

```
      if P1 then
```

```
           A1;
           if Q1 then goto DONE endif;
      endif;
      if P2 then
           A2;
           if Q2 then goto DONE endif;
      endif;
        .
        .
        .
DONE: stop;
```

The GOTO removal algorithm will express this as nested IF-THEN-ELSE statements without GOTOs. However, the depth of nesting increases with the length of the chain. Since each nested IF-THEN-ELSE has two branches, the total size of the compound statement grows exponentially with the chain length.

In an application involving several (~ 10-20) COBOL programs of tens of thousands of lines each, this difficulty occurred with only one program, and within that program only within two segments of code. We adopted the expedient of hand-recoding those segments, using auxiliary Boolean variables to eliminate the duplication. Note that the duplication could also be avoided by introducing dummy loops that execute only once, as discussed by Peterson *et al.* (Peterson *et al.*, 1973), since the GOTOs could be then replaced by loop exits. However, we felt that dummy loops would constitute obscure or misleading code in this case.

Regular Expressions To Programs

Some additional work is required to convert the regular expression representation into a final program that uses conventional constructs such as loops. The reason for this is that the star construct in regular expressions denotes only *completed* cycles in an iteration. The final partial cycle in which the iteration is exited manifests as a subexpression following the loop.

For example, when the algorithm of the preceding section is applied to a flowchart corresponding to the following pseudocode (it is convenient to use C-like notation here):

```
while (true) {a; if P then break; b}
```

the resulting regular expression is `(a -P b)`* `a +P`.

Notice that the conditional-test symbol `+P` occurs in the interior of a sequence. This does not directly match the format of an if-then-else expression, which corresponds to a pattern such as `(+P... | -P...)`, as discussed in the appendix. In general, the regular expression representation can become quite complex, especially when an iteration has several exits. Fortunately, it is possible to convert the derived regular expression back to a program. The main issue that needs to be addressed is that the loops have become partially unfolded in the process of converting to a regular expression because of the final partial cycles. We can undo this damage by refolding the loop, i.e., matching or aligning the expression following the star operator (called the “sequel”) against the body of the star operator, while using the cross-matching of opposing tests to identify the loop exits. For example, when `[a†+P]` is matched against `[a†-P†b]`, this identifies `P` as an exit condition in the middle of the loop.

The match process can be described by a set of recursive reductions, which we summarize here. The reductions are applied in a bottom-up fashion to the regular expression. Thus, inner star-expressions are eliminated before the outer ones. Furthermore, the earlier rules are applied in preference to the later ones. Note that the sole purpose of these reduction rules is to eliminate the star expressions; they are not applicable to ordinary code that does not contain the star operator.

In the following, <X, Y> indicates that the body X is to be matched or aligned with the sequel Y. We use (lp†X) to denote a loop expression involving repeated execution of X until an associated exit statement within X is encountered. For rules with pattern +P...-P, the symmetric rule with -P...+P is understood.

```
%%% Initial Reduction
 (X)* Y                    → (lp <X, Y>)
```

In the remaining reductions, the < , > expressions are understood to be surrounded by the current enclosing lp expression that was introduced by the initial reduction. The exits generated by the reductions are with respect to this current enclosing loop, and are labeled as such in the implemented system. However, for simplicity we have suppressed the labeling in the reductions below.

```
%%% Basic reduction
 <ZX, ZY>                  → Z <X, Y>

%%% Terminal reductions
 <X, X>                    → X
 <+PX, -PY>                → (+PX | -PY exit)
```

Before continuing, we see how these apply to our simple test-case. We have

```
(a -P b)* a +P → (lp <a -P b, a +P>)
               → (lp a <-P b, +P>)
               → (lp a (-P b | +P exit)).
```

The last expression can be rewritten in C-like pseudocode as

```
                 while(true) {a; if (!P) b else break;}
```

which is equivalent to the original program.

The remaining reductions deal with more complex matches involving conditionals and loops.

```
%%% Reductions for disjunctions (bars) and loops without following-code.
 <(+PX | -PY), -PZ>          → (+PX | -P <Y, Z>)        ; left bar
 <+PX, (+PY | -PZ)>          → (+P <X,Y> | -PZ exit)    ; right bar
 <(+PX | -PY), (+PZ | -PW)>  → (+P <X,Z> | -P <Y,W>)    ; double bar
 <(lp X), (lp Y)>            → (lp <X,Y>)               ; inner loop

%%% Eliminate following-code for bars and loops by distributing inside,
 <(+PX | -PY) U, Z>          → <(+PXU | -PYU), Z>
 <X, (+PY | -PZ) U>          → <X, (+PYU | -PZU)>
 <(lp X) U, Y>               → <(lp cover-exits(X,U)), Y>
 <X, (lp Y) U>               → <X, (lp cover-exits(Y,U))>
```

The cover-exits(Y,U) function places a copy of U at each exit point within Y. (Thus, the following-code is distributed to each exit.)

For the purpose of constructing flowcharts, we may regard an <X, Y> expression as equivalent to (X | Y exit), although it has an additional computational connotation in terms of focus of attention with respect to the rules. Using this equivalence, and considering the reductions in terms of their effect on the corresponding flowcharts, it is easy to verify that they preserve path-equivalence. Notice that the distribution rules generally duplicate nodes in the flowchart, and the other reductions (apart from the initial one) have the effect of merging equivalent nodes. Actually, the purpose of the distribution rules is simply to facilitate application of the merging rules. Notice that when a distribution rule is used, no further distribution rules are applicable until at least one merging rule follows. Although, for purposes of modularity, we have listed the distribution rules and the bar/loop merging rules separately above, the computational effect is essentially the same as if they were combined to form more general reductions as follows:

```
%%% Reductions for disjunctions (bars) and loops.
<(+PX | -PY) U, -PZ>           → (+PXU | -P <YU, Z>)       ; left bar
<+PX, (+PY | -PZ) U>           → (+P <X,YU> | -PZU exit)   ; right bar
<(+PX | -PY) U, (+PZ | -PW) V> → (+P <XU,ZV> | -P <YU,WV>) ; double bar
<(lp X) U, (lp Y) V>           → (lp <cover-exits(X,U),
                                          cover-exits(Y,V)>) ; inner loop
```

Expressed in this form, it is intuitively clear that the reductions eventually terminate, since each step after the initial reduction strictly reduces the scope of the `<X, Y>` expressions. This is presented more formally in the following theorem.

Theorem 1 *The reduction rules terminate after a finite number of applications.*

Proof: We will associate a non-negative integer, called the *reduction length*, with each `<X, Y>` expression. For definitional convenience, the reduction length of *e*, denoted *RL*(*e*), is defined for *e* ranging over the whole class of expressions formed from “letters” using the concatenation, disjunction, `<.,.>` and `lp` operators. (These are the expressions to which the reduction rules are applied.)

We can define *RL* inductively by the following set of equations.

$$RL\,(\texttt{<X, Y>}) = \max(RL\,(\texttt{X}), 2 + RL\,(\texttt{Y})) \tag{1}$$
$$RL\,(\texttt{(X | Y)}) = \max(RL\,(\texttt{X}), RL\,(\texttt{Y})) \tag{2}$$
$$RL\,(\texttt{(lp X)}) = 1 + RL\,(\texttt{X}) \tag{3}$$
$$RL\,(\texttt{XY}) = RL\,(\texttt{X}) + RL\,(\texttt{Y}) \tag{4}$$
$$RL\,(\texttt{X}) = 1, \text{ if } \texttt{X} \text{ is a “letter”} \tag{5}$$

We now restrict attention to the reduction lengths of only the `<X, Y>` expressions. It can be verified that each of the reduction rules (after the initial one) results in `<X, Y>` expressions (generally one, but possibly two) with strictly lower reduction length than the one to which the rule was applied. It follows that the reduction process must terminate, since no expression can have a negative reduction length. Q.E.D.

Since a reduction step may produce two daughter `<X, Y>` expressions, the total number of reduction steps is bounded by 2^r, where *r* is the reduction length. In practice, it will be much less than this upper bound, since the double bar reduction is the only one that results in two daughter `<X, Y>` expressions. Furthermore, as mentioned earlier, the earlier reduction rules are applied in preference to the later ones. Thus, for example, `<(+PX | -PY) U, (+PX | -PY) V>` will be reduced to `(+PX | -PY) <U, V>` via the basic reduction rule, rather than using distributivity and double bar. We did not encounter significant problems in practice due to the double bar reduction.

The question arises whether the given reductions are adequate for refolding the loops, and restoring the regular expression programs to a conventional form. It seems reasonable to conjecture that this is so, since the reductions are merely undoing duplications that occurred during the construction of the star expressions. Nevertheless, this is still an open question. It may be noted that the reductions appear to cover all the merging situations that can arise within the expression language. Furthermore, experiments involving a judiciously chosen suite of examples confirm in all cases that the output of these reductions is an expression where the `<X, Y>` expressions have been eliminated, and conditionals only occur as in the pattern `(+P... | -P...)`. (Real-world tests involving naturally occurring programs of tens of thousands of lines, carried out under less controlled circumstances, also appear to confirm this.) This is easily converted into a program-like form that combines primitives of the source language using high-level if-then-else and loop constructs. Note that, as pointed out by Peterson *et al.*, the loop construct needs to (and does) allow multilevel exits.

After the reductions are complete, further simplification rules are applied. We only use simplification rules that clearly preserve path-equivalence. For example, one can “factor out” tail code that is common to all exits of a loop, and place it after the loop. Thus, `(lp (+PX | -PY exit))` becomes `(lp†(+PX†|†-P†exit))Y`. (Notice that this has an opposite effect to the distribution rules. However, there is no danger of conflict, since the simplifications are delayed until after the reductions are complete.)

In a typical translation task, further work may be required to translate the “letters” of the language (i.e., simple statements and tests), and to express the high-level constructs in the syntax of the destination language. For example, if the destination language does not allow multilevel exits, these may need to be simulated or finessed by such devices as introducing auxiliary variables, partially unrolling loops, or employing other idioms commonly used for that purpose in the destination language.

An Example

We illustrate the method with an example. Consider the following program fragment, where `A`, `B`, `C`, and `D` stand for arbitrary statements, and `P` and `Q` are arbitrary tests.

```
   L:    A;
   M:    B;
         if P then GOTO L;
         C;
         if Q then GOTO M;
         D;
```

Note that the statement “`if Q then GOTO M`” branches back into the middle of a previous loop. This kind of configuration (overlapping loops) is difficult to translate by hand. The flowchart for this program is the one presented in Figure 2.

As discussed in Section 4, the regular expression form for this program is

```
(A (B -P C +Q)* B +P)*    A (B -P C +Q)*   B -P C -Q    D
```

The first part of the expression shows that there are two loops, with one nested inside the other. Note that there is a “reflection” of the inner loop in the expression following the outer loop. This represents the reprise of the inner loop that occurs during the final partial cycle in which the outer loop is exited.

If we first completely reduce the inner loop star-expressions, we get

```
(A (lp B (-P | +P exit-inner) C +Q))*
 A (lp B -P C (+Q | -Q exit-inner))
D
```

Applying several reductions to the outer loop then leads to

```
(lp A (lp B
            <(-P C +Q | +P exit-inner), -P C (+Q | -Q exit-inner)>
))
D
```

which further reduces to

```
(lp A (lp B  (-P C
                   <+Q, (+Q | -Q exit-inner)>
                                              | +P exit-inner)
))
D
```

and finally to

```
(lp A (lp B (-P C (+Q | -Q exit-inner exit-outer) | +P exit-inner)
))
D
```

which is equivalent to the following program.

```
outer: loop
           A;
inner:     loop
               B;
               exit inner loop when P;
               C;
               exit outer loop when not Q;
           end loop;
       end loop;
       D;
```

Closing Remarks

We have presented a method for restructuring programs by eliminating GOTOs in favor of loops and if-statements. In contrast to previous direct methods, which treat GOTO removal as an isolated problem for programming languages, our technique uses the well-understood relationship between finite state transition networks and regular expressions. This is justified by a linguistic/transformational approach to program semantics (in Appendix A below) that associates transformations from states to sets of states with sets of strings over an alphabet of elementary operations.

The approach has been tested on a suite of examples involving various kinds of loops and jumps, including the example of Ashcroft and Manna (Ashcroft *et al.*, 1972), as well as a program with an irreducible flowchart. It has also been successfully applied to the removal of GOTOs in reverse-engineering IBM assembler (Morris *et al.*, 1996), and commercially to help generate code reports for COBOL programs of several tens-of-thousands-of-lines (Polak *et al.*, 1995).

We conclude that GOTO removal can be successfully reduced to the problem of converting a finite-state transition network to a regular expression, leading to greater conceptual unity. The resulting algorithm is also simpler because the task of recognizing and determining the nesting of loops is disentangled from the task of identifying loop exits. It is gratifying that the algorithm also appears to work well in practical applications.

This work was motivated by a specific need in coping with the reengineering of legacy systems. However, the linguistic/transformational approach to program semantics may have other applications. For example, it could provide the basis of a *program algebra* for manipulating code. It may also provide a bridge between rule based formalisms that are related to finite-state machines and the more conventional forms that occur in programming languages.

Appendix A. Flowcharts and Regular Sets

In this appendix we discuss foundational issues concerning the semantics of the path-derived regular expressions and their relationship to programs. The semantic approach presented here is different from, and simpler than, the standard continuation-based one (Winskel, 1993) that is popular among theoreticians. However, it is well-suited for our purposes.

One might conjecture that the regular set associated with a flowchart in the Peterson *et al.* formulation (Peterson *et al.*, 1973) corresponds to the set of execution traces. Unfortunately, that is not the case.

To illustrate the difficulty, consider the following code fragment where the variable N has *integer* type.

```
for i from 1 to 2*N
    do print 'hi';
```

The flowchart for this fragment is shown in Figure 3.

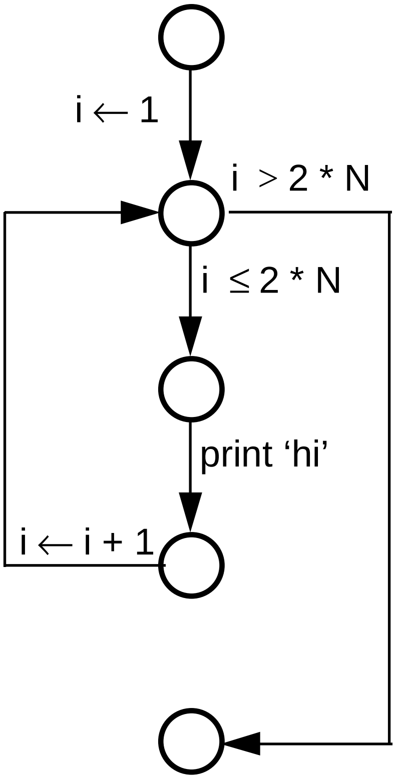


Figure 3. Even number of cycles.

Following the approach of Peterson *et al.*, we can derive a regular expression corresponding to the paths through the flowchart. Assume an alphabet whose “letters” are the primitive statements, and test expressions, that occur in the program. Then the associated regular expression is

$$\boxed{\texttt{i}\leftarrow\texttt{1}}\ (\ \boxed{\texttt{i}\le\texttt{2 * N}}\ \boxed{\texttt{print 'hi'}}\ \boxed{\texttt{i}\leftarrow\texttt{i + 1}}\ )^*\ \boxed{\texttt{i > 2 * N}}$$

where we have indicated the statements and tests that are to be regarded as “letters” by enclosing them in boxes.

Recall that the star operator in a regular expression denotes an arbitrary number of repetitions. However, since 2*N is even, in any real execution of the loop the number of repetitions would necessarily be even. Thus, the precise semantic status of the star operator in the above expression is unclear. In the following paragraphs, we outline a more rigorous treatment of the semantics of such regular expressions in relationship to programs. The basic idea is that the expression is regarded as an algebraic combination of transformations that together determine the total effect of the program.

In this approach, we represent the semantics of a flowchart program (or program fragment) as an operator, or mapping, over states of the world. For technical reasons (specifically, in order to fit both statements and tests into a uniform framework), we consider mappings that transform states into sets of states. These may be thought of as nondeterministic programs. (A deterministic program is then a special case, where the image of each domain element is a singleton set.)

We begin by representing the semantics of the individual "letters," that is, the elementary statements and tests that occur as labels in the flowchart. This approach does not depend on the details of any particular programming language. We assume only that the execution of a statement alters the computer or the world in some unspecified way, and that the evaluation of an elementary test is without side-effects. (The latter assumption is non-essential and could be relaxed by a slightly more complicated treatment.)

Given the above, we assume a set of states, left unspecified, that represents the possible states of the world. A statement A is then associated with a mapping f such that $f(s) = \{s'\}$, where s' is the state of the world that results if A is executed when the world is in state s. A test +P (resp., -P) corresponds to a mapping f such that

$$f(s) = \begin{cases} \{s\} & \text{if } P \text{ is true (resp., false) in } s \\ \varnothing & \text{otherwise} \end{cases} \tag{1}$$

We distinguish two special mappings. The first, called `nop` (for "no-operation"), is the mapping associated with the test `true` that holds in every state. Thus, `nop`(s) = {s} for all s. The second, called `fail`, is the mapping associated with the test `false` that does not hold in any state. That is, `fail`(s) = ∅ for all s.

Next we extend the semantics to apply to strings or words formed from the "letters." We let M_w denote the mapping associated with a letter or string w. If S is a set of states, we write $M_w(S)$ to mean $\cup_{s\in S} M_w(s)$. As usual, Λ denotes the empty string. Then the mapping associated with a string is defined inductively by M_Λ= `nop` and

$$M_{wc}(s) = M_c(M_w(s)) \tag{2}$$

where c is a letter and w is a string. Intuitively, the semantics associated with a string is that of of applying the component "letter" transformations in left-to-right order. Note that it follows from the above definition that if u and v are any strings, then

$$M_{uv}(s) = M_v(M_u(s)) \tag{3}$$

Now we extend the semantics to sets of strings. The mapping M_W associated with a set of strings W is defined by

$$M_W(s) = \bigcup_{w\in W} M_w(s) \tag{4}$$

Note that it follows that $M_\varnothing$= `fail`. Intuitively, the semantics associated with a set of strings is that of treating the individual strings as nondeterministic alternatives. (This is somewhat similar to the situation with a set of clauses in the nondeterministic language Prolog, except in Prolog the order of the clauses is significant.)

We can further extend the semantics to formalisms such as finite-state transition networks and regular expressions that specify sets of strings, assuming an alphabet of elementary statements and tests. Thus, the semantics of regular expressions such as $(E_1 \mid E_2)$ or E^* is interpreted in terms of the semantics of the specified regular sets. This implies that $(E_1 \mid E_2)$ combines E_1 and E_2 as nondeterministic alternatives, while E^* corresponds to the infinite family of alternatives $(\Lambda \mid E \mid EE \mid \ldots)$.

Finally, the identification of flowchart programs with finite-state transition networks extends the semantics to ordinary program constructs that can be expressed by flowcharts. For example, “if Test then Action1 else Action2” translates as

```
(+Test Action1 | -Test Action2).
```

Similarly, “while Test do Action” becomes

```
(+Test Action)* -Test
```

These appear to be reasonable renderings of the intuitive semantics.

In light of the above, the regular set associated with a flowchart is seen not as a set of possible traces, but as something quite similar: a set of straightline programs that are alternatives in a nondeterministic sense.

Consider once again the expression

$$\boxed{i \leftarrow 1}\ (\ \boxed{i \le 2 * N}\ \boxed{\texttt{print 'hi'}}\ \boxed{i \leftarrow i+1}\ )^*\ \boxed{i > 2 * N}$$

where N is of type integer. The set of strings denoted by this does indeed include strings where the repetition occurs an odd number of times. However, these represent “null” mappings, i.e., mappings that transform every state to the empty set of states. Thus, they contribute nothing to the combined mapping. Intuitively, they are nondeterministic alternatives that fail.

Appendix B. Sample Programs

Note: The input and output format in these samples has been edited for readability, but the structure is as produced by the implemented system.

Before GOTO removal **After GOTO removal**

Program1: Contains inaccessible code

```
  <<lb1>>:                            loop
    DISPLAY("READY");                     DISPLAY ( "READY");
  <<lb2>>:                                DISPLAY ( INPUTGANGMSG);
    DISPLAY(InputGangMsg);                ACCEPT ( MASSGANG, CARDREADER);
  <<lb3>>:                                if VALIDGANG then exit endif;
    ACCEPT(MassGang, CardReader);         DISPLAY ( "INVALID GANG")
  <<lb4>>:                            endloop
    goto <<lb6>> if ValidGang;
    DISPLAY("INVALID GANG");
    goto <<lb1>>;
    goto <<lb7>>;
  <<lb6>>:
    goto <<lb5>>;
  <<lb7>>:
  <<lb5>>
```

Program2: Tests in middle of loops. i.j reach n.

```
    set sum = 0;                   set SUM = 0;
    set i = 1;                     set I = 1;
  <<lp1>>:                         loop1
    set j = 1;                        set J = 1;
  <<lp2>>:                            loop2
    set sum = (sum + c(i,j));            set SUM = (SUM +  C ( I, J));
    goto <<out2>> if (j == n);           if (J = N) then exitloop2 endif;
    set j = (j + 1);                     set J = (J +  1);
    goto <<lp2>>;                     endloop2
  <<out2>>:                           if (I = N) then exitloop1 endif;
    goto <<out1>> if (i == n);        set I = (I +  1);
    set i = (i + 1);              endloop1
    goto <<lp1>>;                 print (SUM);
  <<out1>>:
    print(sum)
```

Program3: Multilevel exit when c = 0.

```
  set sum = 0;               set SUM = 0;
  set i = 1;                 set I = 1;
<<lp1>>:                     loop1
  set j = 1;                    set J = 1;
<<lp2>>:                        loop2
  set c = c(i,j);                 set C = C ( I, J);
  goto <<out1>> if (c == 0);      if (C == 0) then exitloop1 endif;
  set sum = (sum + c);            set SUM = (SUM +  C);
  set j = (j + 1);                set J = (J +  1);
  goto <<lp2>> if (j <= n);       if not(J <= N) then exitloop2 endif;
  set i = (i + 1);              endloop2
  goto <<lp1>> if (i <= n);     set I = (I +  1);
<<out1>>:                       if not (I <= N) then exitloop1 endif;
  print(sum)                 endloop1
                             print (SUM);
```

Program4: Ashcroft and Manna program

```
  A ();                     A ();
<<lp1>>:                    loop1
  goto <<s1>> if P;            if not P then
  goto <<s2>> if Q;               if not Q then
  G ();                              G ();
  goto <<finish>>;                   exitloop1
<<s2>>:                           else
  B ();                              B ();
<<lp2>>:                             loop2
  goto <<s3>> if R;                     if not R then exitloop2 endif;
  goto <<s4>> if S;                     D ()
  F ();                              endloop2;
  goto <<finish>>;                   if S then
<<s1>>:                                 C ()
  E ();                              else
  goto <<lp1>>;                         F ();
<<s3>>:                                 exitloop1
  D ();                              endif
  goto <<lp2>>;                   endif
<<s4>>:                        else
  C ();                           E ()
  goto <<lp1>>;                endif
<<finish>>:                 endloop1
  halt ()                   HALT ()
```

Program5: Program with an irreducible flowgraph

```
  goto <<e1>> if "red-first";   if "red-first" then
  goto <<e2>>;                      loop
<<lp>>:                                 PRINT ( "red");
  goto <<finish>> if "i=0";             PRINT ( "green");
<<e1>>:                                 EXECUTE ( "i--");
  print("red");                         if "i=0" then exit endif
<<e2>>:                             endloop
  print("green");               else
  execute("i--");                   loop
  goto <<lp>>;                          PRINT ( "green");
<<finish>>:                             EXECUTE ( "i--");
  execute("halt")                       if "i=0" then exit endif;
                                        PRINT ( "red")
                                    endloop
                                endif;
```

```
                        EXECUTE ( "halt")
```

Bibliography

Ashcroft, E. and Manna, Z. (1972). “The translation of `goto' to `while' programs,” *Information Processing*, **71**, North Holland, Amsterdam, 250–255.

Barrett, W. A. and Couch, J. D. (1979). *Compiler Construction Theory and Practice*, Science Research Associates, Chicago.

Dijkstra, E. W. (1968). ‘Go to statement considered harmful,’ *Communications of the ACM* **11,** 147–148.

Erosa, A. M. and Hendron, L. J. (1994). ‘Taming Control Flow: A structured Approach to Eliminating Goto Statements,’ Proc. IEEE International Conference on Computer Languages, 229–240.

Filman, R. E. (1996). ‘Applying AI to Software Renovation,’ to appear in *Automated Software Engineering*.

Gibbons, A. (1985). *Algorithmic Graph* Theory. Cambridge University Press, New York.

Gibbs, W. W. (1996). ‘Battling the Enemy Within,’ (Technology and Business section, April) *Scientific American*.

Manna, Z. (1974). *Mathematical Theory of Computation*, McGraw-Hill, New York.

Morris, P. and Filman, R. (1996). ‘Mandrake: A Tool for Reserve-Engineering IBM Assembly Code,’ to appear, 3rd Working Conference on Reverse Engineering.

Peterson, W. W., Kasami, T. and Tokura, N. (1973). ‘On the Capabilities of While, Repeat, and Exit Statements,’ *Communications of the ACM* **16,** 503–512.

Polak, W., Bickmore, T. and Nelson, L. (1995). ‘Reengineering IMS databases to Relational Systems,’ Proc. Seventh Annual Software Technology Conference, Salt Lake City, CD-ROM.

Ramshaw, L. (1988). ‘Eliminating go to's while Preserving Program Structure,’ *Journal of the ACM* **35**, 893–920.

Reasoning Systems. (1990). *REFINE User’s Guide, Version 3.0*, Reasoning Systems Inc, Palo Alto, California.

Winskel, G. (1993). *The Formal Semantics of Programming Languages*, MIT Press, Cambridge, Massachusetts.

Wulf, W. A., Shaw, M. and Hilfinger, P. N. (1981). *Fundamental Structures of Computer Science*, Addison-Wesley, Reading, Massachusetts.

Authors

Paul Morris received his undergraduate education at University College, Cork (where George Boole once taught) and obtained Ph.D.’s in Mathematics (1974), and Computer Science (1984) from the University of California, Irvine. After completing his education, Dr. Morris joined IntelliCorp, where he worked on the KEE™ expert system shell and related projects. More recently, he has consulted with Lockheed-Martin on analysis and reengineering of legacy software. Dr. Morris has published on truth maintenance, planning, knowledge representation, constraint satisfaction and software reengineering. He was the recipient of a Best Paper Award at the National Conference on Artificial Intelligence in 1987.

Ronald A. Gray is an expert on the understanding and renovation of computer systems through inferential methods. He has worked for Lockheed since 1986, reengineering systems such as the military’s MICAP logistics system, Theater High-Altitude Air Defense simulations, neural network simulations, and avionics and operating systems software in general. Prior to coming to Lockheed, he renovated software for Hughes Aircraft Company and First Chicago Corporation. He received his B.S. in Physics from The University of Miami, and his M.S. in Theoretical Physics from the University of Chicago, where he was a National Science Foundation Fellow at the Enrico Fermi Institute for Nuclear Studies.

Robert E. Filman is a Chief Technologist of the InVision software reengineering project in Lockheed Martin’s Software Technology Center. Prior to coming to Lockheed, he worked at IntelliCorp, Hewlett-Packard Laboratories, and Indiana University, Bloomington. He has published in the areas of software engineering, distributed computing, network security, programming languages, artificial intelligence, and human-machine interface. Dr. Filman received his B.S. (Mathematics, 1974), M.S. (Computer Science, 1974) and Ph.D. (Computer Science, 1979) from Stanford University.